Implementing Network Partition-Aware Fault-Tolerant CORBA Systems

Stefan Beyer, Francesc D. Muñoz-Escoí and Pablo Galdámez
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
Spain
{stefan, fmunyoz, pgaldamez}@iti.upv.es

***Abstract*— The current standard for Fault-Tolerance in the Common Object Request Broker Architecture (CORBA) does not support network partitioning. However, distributed systems, and those deployed on wide area networks in particular, are susceptible to network partitions.**

The contribution of this paper is the description of the design and implementation of a CORBA fault-tolerance add-on for partionable environments. Our solution can be applied to an off-the-shelf Object Request Broker, without having access to the ORB's source code and with minimal changes to existing CORBA applications.

The system distinguishes itself from existing solutions in the way different replication and reconciliation strategies can be implemented easily. Furthermore, we provide a novel replication and reconciliation protocol that increases the availability of systems, by allowing operations in all partitions, including non-majority partitons to continue.

I. Introduction

The Common Object Request Broker Architecture (CORBA) [1] originally did not provide fault-tolerance. The Fault-Tolerant CORBA specification (FT-CORBA) [2] has been added to introduce a degree of fault-tolerance. However, one of the biggest draw-backs of FT-CORBA, is that it does not consider network partition failures. As distributed systems become more and more used in wide-area networks, communication link failures are likely. Link failures might split a network into two or more partitions which are isolated from each other. Replication techniques reduce the likelyhood of a resource being unreachable. Nevertheless, updates to object states on isolated nodes can lead to inconsistencies, as not all replicas of the object might be reachable for synchronising this new object state. To deal with this consistency problem, replication and reconciliation protocols are required. These protocols are very application-specific, since a certain degree of inconsistency might be acceptable for one application, but not for another. Similarly, certain problems, such as nested invocation might simply not arise in some applications. Existing solutions to add partition-aware replication to CORBA do not allow these application-specific replication policies to be configured easily.

The main contributions of this paper are twofold: Firstly, we present the architecture of a middleware add-on that adds fault-tolerance to CORBA in a partitioned environment by means of replication. The system is part of the *DeDiSys* project [3]. *DeDiSys* aims at providing fault-tolerance through add-ons for various middlewares. In contrast to other systems, the modular design of the *DeDiSys* replication support allows different replication and reconciliation policies to be provided easily. The design of the replication support is based on a separation of mechanism and policy. We extract replication policy from the replication manager and place it into a replication protocol component. The replication manager and the replication protocol components provide fixed interfaces. New replication protocols can be implemented by replacing the replication protocol component.

The second contribution of this paper is a replication and reconciliation protocol, which allows consistency to be traded for higher availability in a controlled manner. The Primary per Partition Protocol (P4) is the default protocol in our architecture. The protocol allows consistency to be traded for availability in a configurable manner. Objects in all partitions of a partitioned systems are allowed to continue serving requests, including write requests. The state of replicas of the same objects in different partitions may diverge in some cases. Whether or not possible state convergence is allowed for a certain object can be configured by the application programmer. To do so, we base consistency on integrity constraints, which a programmer can use to define restrictions on the state of a single object or between several objects. These constraints can be marked as critical for those objects for which strict consistency is required; that is, those objects, whose state should not diverge in different partitions.

In a system, which allows the state of replicated objects in different partitions to diverge, protocols for restoring consistency at the time partitioning is repaired are required. To this end, we also introduce a reconciliation protocol, which can restore consistency either fully automatically, or with the help of the application, depending on how the protocol is configured. A general description of the P4 protocol together with a analytical evaluation has been published in [4]. In this paper we describe a revised version of the protocol and its actual implementation in our CORBA architecture.

The rest of this paper is organised as follows: In Section II we present realted work in the field of fault-tolerant CORBA architectures. In Section III the general architecture of the DeDiSys middleware add-on is described, together with the employed system model. This is followed by a description of the mapping of the general DeDiSys architure to the CORBA middleware and the actual implementation in Section IV. In Section V we introduce our new replication and reconciliation protocol. Finally, we conclude the paper with a description of future work and a conclusion in Section VI and Section VII respectively.

II. Related Work

In order to add fault-tolerance to CORBA, certain mechanisms, such as replication, are required. Existing systems either implement the FT-CORBA [1] standard to provide fault-tolerance or suggest their own fault tolerance extensions. Some systems reviewed here were simply developed before the standard was defined. Other systems try to overcome some of the drawbacks associated with FT-CORBA. As *DeDiSys* is a research project, aimed at partitionable distributed systems, which are not covered by the FT-CORBA standard, we do not consider FT-CORBA compliance as the main factor for this review.

In literature, approaches to add fault-tolerance mechanisms to CORBA are typically classified into three categories: In the **integration approach**, the ORB itself is modified to include the required fault tolerance mechanisms. It is easy to provide transparency using this approach, but existing commercial ORBs cannot be used. Orbix+Isis [5], Electra [6] and Maestro [7] are examples of systems using the integration approach. More recently, the authors of [8] and [9] have proposed the integration of group communication support by modifying the CORBA Open Communication Interface (OCI) and using the Pluggable Protocols Framework [10]respectively.

In the **service approach**, the mechanisms required to provide fault tolerance are provided as CORBA services. This approach has the advantage that existing ORBs can be used. However, transparency is difficult to achieve with this approach, as applications have to be aware of the fault tolerance services. Object Group Services (OGS) [11] and Newtop Object Group Service [12] provide services for object group support which can be used to provide fault-tolerance. FTS [13], OPEN EDEN [14], IRL [15] and AQuA [16] are examples of reliable CORBA systems using the service approach, although it can be argued that these systems also use elements of the interceptor approach.

In the **interceptor approach**, CORBA invocations are intercepted and redirected to fault tolerance mechanisms. Recent systems make use of CORBA Portable Interceptor [1]. The only systems using a pure interception approach we are aware of are Eternal [17] and DAISY [18].

Three of the systems mentioned above - Maestro, FTS and Eternal - provide some support for network partitioning. Therefore, these systems are reviewed here in more detail. Newtop also provides support for network partitioning, but, as a mere object group toolkit, does not provide any support for reconciling replica state after partitioning. Therefore, we do not discuss Newtop in detail here.

Maestro uses the integration approach. The system was developed before the FT-CORBA specification existed. It is not a pure CORBA implementation, but was designed as a distributed object layer to be used on its own or to be integrated in CORBA or in other distributed object technologies. Only updates in one partition are permanently accepted, but in contrast to the regular primary partition model [19], the decision on which partition dominates is postponed until recovery time. At recovery time the partition with "the most updated" state is chosen.

FTS is an attempt to remain close to the FT-CORBA specification, whilst also providing support for partitioning. The system uses a mixture of the service and interceptor approaches. A group object adapter (GOA) is provided as a CORBA object adapter. The GOA is implemented on top of the portable object adapter (POA) to allow for object groups. In *DeDiSys* we use also use the idea of an object adapter providing object group support. FTS uses the primary partition model for consistency in case of network partitioning.

Eternal is probably the most advanced of the systems of which we are aware in terms of support for partitioning, despite being one of the oldest systems. The system allows for active and passive replication. The Eternal replication manager keeps track of replicated objects. CORBA messages are intercepted at the transport level and are redirected using the Totem group communication toolkit [20]. As far as we know, Eternal is unique in partition-aware CORBA systems, in that it does not use a variant of the primary partition model, but does allow operations in all partitions to continue. A simple reconciliation algorithm is provided. Conflicts that cannot be resolved are reported to the application.

In *DeDiSys* we make use of some techniques from Eternal, DAISY and FTS. In particular, we use interception, as in Eternal and DAISY, and the implementation of the replication support as a CORBA object adapter, as in FTS. In contrast to Eternal's interception at the operating system level approach we use DAISY's approach of using portable interceptor, which were not available when Eternal was designed. Furthermore, in *DeDiSys* we aim at making replication and recovery flexible and configurable. To this end we do not embed replication protocol and reconciliation policy in the replication manager, as done in Eternal, but provide an easily interchangeable replication and reconciliation protocol component.

III. The *DeDiSys* Architecture

A. Overview

The *DeDiSys* project aims to introduce fault-tolerance in distributed object systems through replication. In replicated systems, there is a trade-off between availability and consistency, as it is difficult to maintain consistency between replicas without restricting certain object accesses when parts of the distributed system are inaccessible. Consistency is particularly difficult to maintain, if network link failures are to be tolerated

and all parts of partitioned system are to continue operating. In *DeDiSys* we aim at the trade-off between availability and consistency to be configurable. The architecture is designed as a add-on for various middlewares, in which different replication and reconciliation techniques can be implemented. This paper focuses on the design and implementation of the CORBA add-on.

The main idea behind DeDiSys is to base consistency on integrity constraints. Integrity constraints are present in most applications, but are usually implicitly hard-coded in the software. In DeDiSys integrity constraints can be expressed explicitely and are evaluated by a Constraint Consistency Manager (CCM). The CCM is not the subject of this paper. It has been described previously in [21]. In this paper we assume the presence of such a CCM component, which allows constraints to be registered and evaluated. Integrity constraints are associated with object methods, in the form of pre- and post-conditions, and with sets of objects. We distinguish two priorities for constraints. A constraint can be marked *critical* or *regular*. Replication protocols can make use of these priorities to temporarily allow certain inconsistencies.

The P4 replication protocol is the default replication protocol in the DeDiSys architecture. This novel protocol allows objects in all partitions in a partitioned system to be updated. Operations are rejected when a constraint is not met or cannot be evaluated. Furthermore, constraints the protocol makes use of the priorisation of constraints. The consistency of regular constraints can be relaxed during the presence of a failure in the system. That is, it is possible to evaluate non-critical constraints using possibly out of date object states. Critical constraints on the contrary can only be evaluated on up-to-date data. Different replicaton protocols might make different use of the priorisation of the constriants or not use it at all. In theory it would be easy to extend the architecture to provide more than two constraint priorities, but since all the replication protocols that have been implemented so far only use the two priorities, we have not done so.

B. System Model

The "crash model" [22] is assumed for node failures, and the "link failure model" [23] for communication services. As we cannot distinguish between a failed node and an isolated node until recovery time, we treat every failure as partitioning.

In order to provide support for partitioning, *DeDiSys* uses the Spread group communication and membership toolkit [24]. Spread provides the extended virtual synchrony model [25]. This model simplifies the reconciliation process of potential replication protocols, as nodes are aware which views have been installed in re-joining partitions.

Furthermore, we assume the presence of a transaction manager that provides nested transaction. DeDiSys provides such a transaction manager, which is based on the group membership service, rather than on time-outs as other transaction managers do. However, a detailed description of the transaction manager is outside the scope of this paper.

We employ the *passive replication* model in our default replication protocol, although the architecture can easily be extended to deal with active replication. In passive replication [26] [27] requests are only processed by one *primary copy*. Updates are then propagated to the *secondary copies*. The passive model lends itself to a system where consistency is to be configured as it allows variations in the way updates are propagated. If *synchronous* update propagation is used, a primary copy must propagate any updates immediately; that is, before the result of the operation that has caused the update is returned to the client. In *asynchronous* update propagation the result is returned and the propagation of state changes performed some time later. We leave the choice of which update propagation paradigm to use to the replication protocol, although our default protocol uses synchronous update propagation.

IV. CORBA Implementation

A. Modular Design

In order to allow different replication strategies to be implemented as easily as possible, we have based our design on a **separation of mechanism and policy**. Replication mechanisms are basic primitives, such as the ability to create a replica or manage the relation between object references and replica references. The provided mechanisms can be used in different ways to implement replication policies, such as the object state transfer policy or the reconciliation strategy. Policies may vary, whereas mechanisms are provided to support different policies.

In conventional systems policies and mechanism are often embedded in the same component. This makes it difficult to implement different policies. In *DeDiSys*, we extract replication and reconciliation policy from the main replication component, which only provides mechanisms that allow to implement a variety of policies.

As *DeDiSys* is a middleware add-on, it is one of our design goals to allow it to be integrated easily in an existing CORBA environment. We use interception of CORBA invocations to redirect call through *DeDiSys*, in order to allow the integration of *DeDiSys* with an off-the-shelf ORB, without having to modify the ORB.

Replication should be as transparent to the application as possible, to avoid having to apply many changes to existing applications. The integration of *DeDiSys* on the client side is completely transparent. That is, the client is not aware it is dealing with a replicated object and existing CORBA clients do not have to be modified to use *DeDiSys*. However, the server application should have some control over the replication support. Therefore, a simple interface provides mechanisms, such as replica creation, and has to be used by the server application. It is our goal to make CORBA server applications as easy to port to *DeDiSys* as possible, whilst allowing configurability of key parameters, such as number and location of replicas.

B. Implementation

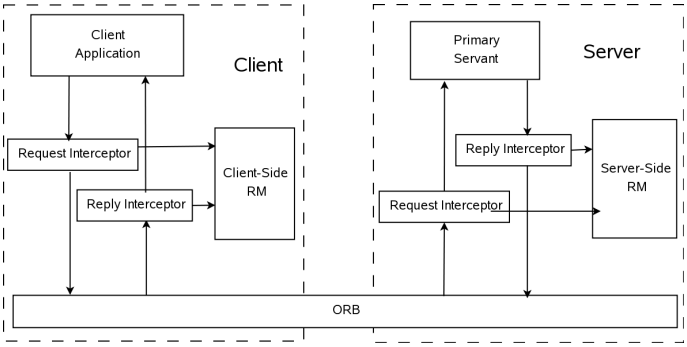

Fig. 1. Replication Support Overview

1) Overview: Figure 1 shows how the *DeDiSys* replication support is integrated in CORBA. Portable interceptor are used to transfer control to the replication support, without client code having to be modified. The client invokes an object in the standard CORBA way, using a logical object reference. The *DeDiSys* replication support takes care of identifying the real object reference of the primary replica. The **client-side request interceptor** is used to intercept object invocations, before they are sent. This interceptor uses the replication suport to obtain the reference of the primary replica and redirects the invocation to this primary replica. The replication manager is also used to trigger some replication protocol specific tasks that might need to be executed before the invocation can begin.

On the server side, the **server-side request interceptor** also intercepts the incoming request, in order to trigger replication protocol specific tasks. The object invocation is then executed in the standard CORBA way. Before the result is returned to the client, control is again passed to the replication support. At this stage the replication protocol might require changes in the accessed object's state to be propagated to the secondary replicas of the object.

Before the request is delivered to the client application the reply is again intercepted on the client side by the **client-side reply interceptor**. At this stage a replication protocol might trigger consistency checks that could cause the invocation to be undone.

C. Object Reference Management

We distinguish between **logical object references** and **replica references**. When using the term logical object reference, we are referring to an object, which might have various replicas, rather than a specific replica. When using the term replica reference we are referring to the actual reference of an object replica; that is, a reference of a real CORBA implementation of a logical object. Only logical object references are visible to client applications. The replication support keeps track of which logical object references are associated with which replica references and redirect invocations on logical object references to the correct replica object reference.

Each object is also given a unique system wide string name. Each object name corresponds to exactly one logical object reference, but to several replica references.

D. System Modes

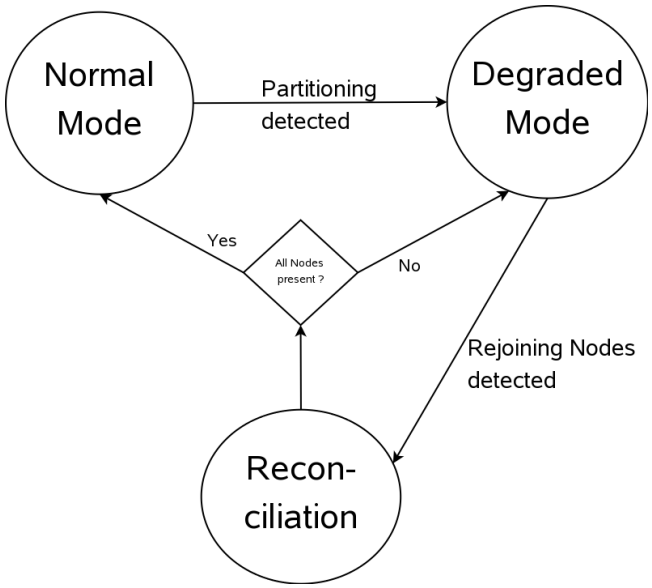

Fig. 2. P4 System Modes

During operation we distinguish between three system modes. These modes and the transitions between them are pictured in figure 2. In *normal mode* there are no detected faults present in the system. The membership view on each node contains all the nodes in the system; that is, the all the nodes that the system is configured to include. When a membership view change is detected by the underlying group membership service, the system moves into degraded mode. In degraded mode the replication protocol might have to take different actions. For instance, certain operations might have to be rejected or version numbers of updated objects might have to be saved to a data-structure. When the group membership service reports one or more or joining or re-joining nodes, the system enters *reconciliation mode*. Control is passed to the replication protocol which in turn can invoke a reconciliation protocol sub-component. A replication protocol might reject, block or accept operations during this mode, depending on how it deals with the consistency issues that arise with accepting state changes during the resolution of possible inconsistent states. When reconciliation is complete the system either returns normal mode or to degraded mode, depending on whether all faults have been repaired or whether some nodes are still not in the new membership view.

E. Components

1) The Replication Manager: The central component of the DeDiSys replication support is the replication manager (RM), which is shown in Figure 3. The main task of the RM is to keep track of objects and their replicas. Several tables are maintained, that map logical object references and object

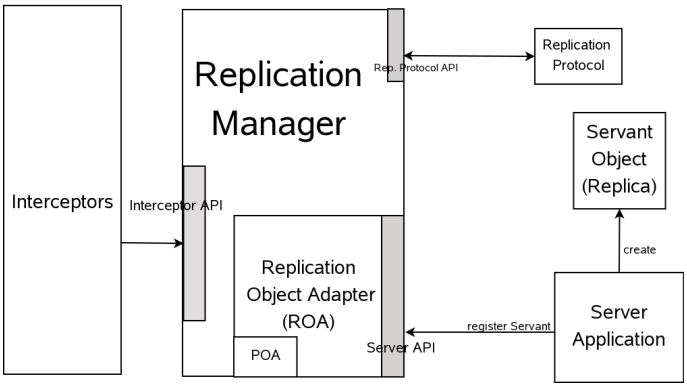

Fig. 3. Replication Support Overview

names to replica references, replica references to nodes and vis versa. These tables are used indirectly by the replication protocol to re-direct invocations to the corresponding primary replica of an invoked object.

The RM consists of various sub-components. Only the **Replication Object Adapter (ROA)** is visible to the server application. The ROA is a CORBA object adapter. It internally uses CORBA's Portable Object Adapter (POA) and provides standard POA functionality, such as associating objects with object references. In addition, it manages object replicas and allows replicas to be created and associated with a logical object.

The RM is an "application" of the Spread group membership and communication service. RMs on different nodes use Spread to exchange information on new replicas or to broadcast replica role changes. Furthermore, the RM keeps track of which nodes are reachable. Therefore Spread callbacks handling the reception of group messages and new membership views need to be implemented in the RM. This infomation is used to set the system mode. If a membership view indicates that not all nodes are currently reachable, the RM enters degraded mode and notifies all system components of this new system mode. Similarly, when the group membership service indicates joining or re-joining nodes, the RM is responsible for triggering reconciliation and enter normal mode, when reconciliation is complete and the system is fully recovered, or re-enter degraded mode, when reconciliation is completed and further nodes remain unreachable.

The RM also interacts with a **replication protocol** component (RP), in which replication protocol details, such as update transfer policies, are implemented. By encapsulating such policy in a separate component with a defined interface, the replication protocol can be changed easily.

Furthermore, the RM serves as a communication interface for other components. Although in theory, each component can implement handlers for the reception of group communication messages, the RM centralises communications. A type field in our messages indicates which component a message should be delivered to and the RM used this field to re-direct messages to their corresponding components. Similarly, components use the RM to transmit messages.

2) Replication Manager APIs: The RM provides three different APIs:

- The **Interceptor Interface** serves as an entry point for the interceptors. As described in Section IV-B.1 there are four interceptors. These interceptors use the RM's interceptor API to pass control to the RM before and after each operation. This is done on the client side, as well as on the server side. The RM in turn passes control to the RP. Not all replication protocols will require action to be performed in each of these interception points, but they are given the chance to do so. In fact, our default replication protocol does not make use of all four interception points.
- The **Server API** is provided by the ROA. As explained above, the ROA performs the functionality of the standard POA. Therefore, many methods of this API form part of the standard POA API. In addtition, the server API allows logical object references to be created. Replicas can be created and associated with logical object references. Furthermore, existing logical object references can be discovered to associate further replicas with the same logical object reference. Finally, as each object has an unique string name, the server API, also provides basic name server facilities.
- The **Replication Protocol API** gives the RP to controlled access to the RM internal tables managing objects and theire replicas. Operations to obtain the primary replica reference of an object, to promote a secondary object to a temporary primary and to discover all the secondary copies of an object for update propagation are provided.

3) The Replication Protocol Component: The replication protocol (RP) component encapsulates replication and reconciliation policies. We locate replication policy and reconciliation policy in the same component, as the policies have to match each other. For instance, a replication protocol that allows updates in each of the partitions of a partitioned system requires a reconciliation policy that allows the system to recover from the inconsistencies this might introduce.

The RM passes control to the RP before and after every object invocation, in order to allow the RP to allow or deny certain object invocations to maintain consistency and to keep track of changes to objects and maintain internal data structures that hold information necessary for reconciliation. The activities of the RP in a healthy system vary from that in a system in which one or more nodes are not reachable, as different data-structures have to be maintained in these different system modes. Furthermore, the RP implements update propagation and reconciliation.

Different RPs can be implemented by modifying the RP component. To this end, the RP component consists of an abstract `ReplicationProtocol` class, which should be extended, in order to implement a replication protocol. The `ReplicationProtocol` class also provides default implementations of some methods, that may or may not be overwritten by a particular replication protocol. A default

update propagation method is provided. The method can be called by any subclass implementing a specific replication protocol to broadcast the state of a specific primary replica to all secondary copies. Furthermore, a default method handling incoming replica updates is provided. This method just sets the state of all the secondary copies it holds of a particular primary copy to that included in the message. Both the update propagator and the incoming message handler can be overwritten by protocols that require more specialised implementations.

In the next section, we will describe how we have used this modular architecture to implement a novel replication protocol that bases consistency on integrity constraints.

V. The Primary Per Partition Protocol

A. Overview

The primary per partition protocol (P4) is a replication protocol that allows consistency to be traded for availability. The protocol uses passvie replication. That is, invocations are direted to a primary replica. Once the invocation has completed, possible object state changes are propagated to all secondary copies of the object.

In the P4, when a primary copy of an object is not available due to a node failure or network partitioning, a reachable secondary copy is promoted to a temporary primary and invocations are allowed to proceed on this temporary primary. In contrast to other solutions, we allow the state of temporary primaries of a single object to be modified in all the partitions of the systems. This can cause the state of replicas to diverge. To deal with this inconsistency a reconciliation protocol is executed when partititons merge, in order to restore data consistency.

The protocol makes use of integrity constraints, as explained in Section III. If an operation causes a constraint to be violated it is rejected. Furthermore, the degree to which inconsistencies are allowed can be configured using the priorisation of integrity constraints in DeDiSys. In addition to association of constriants with operations in the form of pre- and post-conditions and with sets of objects, constraints in DeDisys are classified as **critical** or **regular**, as explained in Section III. Using the P4, critical constraints can only be evaluated, if all the objects they refer to are known to be up to date. That is, all primary copies must be reachable. Regular constraints can be evaluated on secondary copies of objects for which the primary is not reachable. Such a copy might hold a stale state. We call this a consistency threat.

An example of typical integrity constraint used in this way is a constraint placed on a bank account object. The constraint states that the balance of the account cannot go below zero. The constraint is associated with a withdrawl operation as a post-condition. Should the system be partitioned and the freshness of the account balance value cannot be verified when a customer tries to withdraw money, a consistency threat is present, as the constraint cannot be evaluated on a value known to be up-to-date. The bank might wish to never allow this possible inconsistency, in which case, the constraint is marked as critical. However, for some important customers the bank might be willing to take a risk. In this case, the constraint is marked as regular and consistency threats are accepted by our protocol. This constraint involves only one object. However, constraints can involve several objects. An example of such an inter-object constraint is a constraint restricting the sum of all bank accounts a single customer might have with a bank to be above zero.

Consistency threats are resolved at reconciliation time. A post-condition expressed as a regular constraint has to be re-evaluated, once all the objects are up-to-date. However, a pre-condition is not re-evaluated at reconciliation time. Replica conflicts and data integrity can be dealt with automatically or referred to the application, depending on the chosen reconciliation strategy.

B. Protocol Details

This section describes the behaviour of the P4 in different system modes. The protocol makes full use of the replication managers's three system modes and operates differently in each mode. As explained above, the protocol is triggered using the RM's interceptor interface. To send update propagation messages and receive such messages the default implementations in the `ReplicationProtocol` super-class are used. This particular implementation of the protocol also makes use of the DeDiSys transaction manager. Invocations are executed within nested transactions, which are started and ended by the replication protocol. Transaction management has been omitted in this section in order to simplify the protocol description, but it is important to be aware of its existence, as it explains how rollbacks are performed.

Read operations cannot introduce data inconsistencies. The following description therefore focuses on write operations. Read operations are treated in the same way, but no object state changes occur.

Normal Mode

1) All object invocations are directed to the primary replica.
2) All the pre-condition constraints, associated with the operation are evaluated. If a constraint is not met, the operation is aborted.
3) The operation is invoked. Nested invocations might be started.
4) Once the primary replica has updated its local state, all the post-condition constraints, associated with the operation are evaluated. If a constraint is not met, the operation is aborted.
5) Once these checks have been successfully completed, all primary replicas updated in the operation propagate the new object states to the backup replicas.
6) Once this update transfer has terminated, the operation result is returned to the client.

Degraded Mode

A write operation in degraded mode is similar to that in normal mode with the following additions:

1) If the primary copy of an object being written to is not found, a secondary copy is chosen in some predetermined way, for example based on the replica iden-

tifier. The chosen secondary replica is promoted to a "temporary primary". This is not done, if the operation has a critical constraint as a pre- or post-condition.

2) Objects that are changed are marked as "revocable", if any of the post-condition constraints associated to the operation that has been executed has been evaluated on possibly stale objects.
3) Critical constraints are not evaluated, if a participating object might be stale. If this is the case, the operation is aborted.
4) Regular post-condition constraints with possibly stale objects are marked for re-evaluation at reconciliation time.
5) Operations with critical constraints that include a revocable object are not permitted, so that critical constraints cannot be violated retrospectively, when a revocable object is rolled back.

Reconciliation

When two or more partitions re-join, reconciliation is started. During this process no write operations are processed. Reconciliation is done in three phases:

Phase 1: Restoring replica consistency.

When partitions are being joined, replica consistency is restored. If two primary copies of the same object have been modified in different partitions a write-write conflict has occurred. To solve this conflict the application is asked to resolve the conflict. To this end a handler routine which has been previously registered by the application is called. Conflict resolution strategies the application may employ range from choosing one of the conflicting primary copies to installing a completely new version.

Phase 2: Restoring constraint consistency.

All constraints that are marked for re-evaluation and for which the original primary copy of all participating objects is now available are now re-evaluated. If a constraint is violated, the application is again asked to resolve the conflict. To this end another handler routine is called. The application handler can restore consistency by setting one of the objects marked as revocable to a state that meets the constraint. All other post-condition constraints of operations that have been executed during partitioning and in which the revocable object participates have to be re-evaluated. This re-evaluation has to be performed to avoid constraints being violated retrospectively.

Phase 3: Updating secondary copies.

Finally, all changes to primary copies which have occurred in phase 1 or phase 2 have to be applied to the secondary copies of the modified objects.

C. Automatic reconciliation

The P4 protocol can also be employed without application interaction; that is, through using automatic reconciliation, instead of application handler routines. However this involves storing a large amount of extra data about object changes during degraded mode. For each object, a list with previous versions has to be kept. Furthermore, for each of the object versions a list of the nodes present in the current partition during the time of the write access needs to be kept. Finally, for each regular post-condition constraint, a reference to the last known version to meet the constraint of each updated participating object needs to be saved.

If several partitions try to re-join and write-write conflicts between two primary copies occur, the protocol can choose one of the primaries according to some pre-defined precedence order or a more complex algorithm.

Constraint violations that are detected at reconciliation time could also be dealt with automatically. If on re-evaluation a regular constraint is not met, one associated object marked as revocable is chosen to revert to its previous version. Among all the revocable objects, one of them is chosen following an increasing order of object identifiers. This object is reverted to previous versions repeatedly, until a version is found that either leads to the regular consistency constraint being met or has the same version number as the last known version that satisfied the constraint. If the later case occurs, without the constraint being met, another revocable object must go through the process of reverting to previous versions. Once the constraint is met, all other regular post-condition consistency constraints associated with the objects that have reverted to a previous version have to be re-evaluated.

VI. Future Work

We are currently in the process of evaluating the architecture. In particular experiments to evaluate the new replication protocol are being performed. To do so, we have implemented a test application which has been designed, so that key parameters, such as the number of constraints, the ratio of critical constraints, the frequency and nesting of invocations and the degree of replication can be easily modified. We are planning to extract these parameters from real work target applications provided by our industrial partners.

Once the suitability of the protocol for the target application has been verified, we will integrate our middleware add-on with these applications to perform further real-world evaluation experiments.

Furthermore, we are planning to improve the current replication protocol and study alternative protocols.

VII. Conclusion

In this paper we have described a fault-tolerance add-on for CORBA. In contrast to most approaches to fault-tolerance in CORBA and the Fault-Tolerance CORBA specification [1], the system can cope with network partitioning. The system forms part of the *DeDiSys* project [3], which aims at providing fault-tolerance add-ons for a variety of middlewares.

The modularity of the design allows different replication strategies to implemented easily. One such strategy, the Primary per Partition protocol has been implemented. The protocol has been found to increase the availability in applications where consistency can be temporarily relaxed, when compared theoretically with the conventional primary partition model

[4]. We are currently evaluating the protocol in practical experiments.

We have implemented the architecture described here in CORBA using Java as an implementation language. We are currently evaluating our implementation with target applications provided by industrial partners of the *DeDiSys* project.

VIII. Acknowledgements

This work has been funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract number 004152).

References

[1] Object Management Group, "The common object request broker architecture (corba) v.3.0.3," March 2004.

[2] ——, "The common object request broker architecture (corba) v.3.0.3. chapter 23. fault tolerant corba," March 2004.

[3] J. Osrael, L. Froihofer, K. M. Goeschka, S. Beyer, F. D. Muñoz-Escoí, and P. Galdámez, "A system architecture for enhanced availability of tightly coupled distributed systems," in *Int. Conference on Availability, Reliability and Security*, 2006, pp. 400–407.

[4] S. Beyer, M. Bañuls, P. Galdámez, and F. D. Muñoz-Escoí, "Increasing availability in a replicated partionable distributed object system," *Lecture Notes in Computer Science: Parallel and Distributed Processing and Applications*, vol. 4330/2006, pp. 682–695, 2006.

[5] IONA and Isis, "An Introduction to Orbix+Isis, IONA Technologies Ltd. and Isis Distributed Sytems Inc." 1994.

[6] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 31–43, 1997. [Online]. Available: citeseer.ist.psu.edu/landis97building.html

[7] A. Vaysburd and K. Birman, "The maestro approach to building reliable interoperable distributed applications with multiple execution styles," *Theor. Pract. Object Syst.*, vol. 4, no. 2, pp. 71–80, 1998.

[8] D. Lee, D. Nam, H. Y. Youn, and C. Yu, "Oci-based group communication support in corba," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1126–1139, november 2003.

[9] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, "Design and implementation of a pluggable fault-tolerant corba infrastructure." *Cluster Computing*, vol. 7, no. 4, pp. 317–330, 2004.

[10] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons, "The design and performance of a pluggable protocols framework for corba middleware," in *IEEE ComSoc TC on Gigabit Networking Sixth International Workshop on Protocols for High Speed Networks VI*, 1999, pp. 81–98.

[11] P. Felber, B. Garbinato, and R. Guerraoui, "The design of a corba group communication service," in *Symposium on Reliable Distributed Systems*, 1996, p. 150.

[12] G. Morgan, S. K. Shrivastava, P. Ezhilchelvan, and M. Little, "Design and implementation of a corba fault-tolerant object group service," in *International Working Conference on Distributed Applications and Interoperable Systems*, June 1999.

[13] R. Friedman and E. Hadad, "Fts: A high-performance corba fault-tolerance service," in *IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems*, 2002, pp. 61–68.

[14] F. Greve, M. Hurfin, and J.-P. L. Narzul, "Open eden: a portable fault tolerant corba architecture." in *Int. Symposium on Parallel and Distributed Computing*, 2003, pp. 88–95.

[15] R. Baldoni and C. Marchetti, "Three-tier replication for ft-corba infrastructures," *Softw. Pract. Exper.*, vol. 33, no. 8, pp. 767–797, 2003.

[16] Yansong (Jennifer) Ren et al., "Aqua: An adaptive architecture that provides dependable distributed objects," *IEEE Trans. Comput.*, vol. 52, no. 1, pp. 31–50, 2003.

[17] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan, "Consistent object replication in the eternal system," *Theor. Pract. Object Syst.*, vol. 4, no. 2, pp. 81–92, 1998.

[18] Taha Bennani et al., "Implementing simple replication protocols using corba portable interceptors and java serialization." in *International Conference on Dependable Systems and Networks*, 2004, pp. 549–554.

[19] A. Ricciardi, A. Schiper, and K. Birman, "Understanding partitions and the "non partition" assumption," in *Workshop on Future Trends of Distributed Systems*, 1993.

[20] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 54–63, 1996. [Online]. Available: citeseer.ist.psu.edu/moser96totem.html

[21] L. Froihofer, J. Osrael, and K. M. Goeschka, "Trading integrity for availability by means of explicit runtime constraints," in *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, 2006, pp. 14–17.

[22] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[23] F. B. Schneider, "What good are models and what models are good?" in *Distributed Systems*, 2nd ed. ACM Press, Addison-Wesley, 1993, ch. 2, pp. 17–26.

[24] Y. Amir, C. Danilov, and J. R. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication," in *International Conference on Dependable Systems and Networks*, 2000, pp. 327–336.

[25] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1994, pp. 56–65. [Online]. Available: citeseer.ist.psu.edu/moser94extended.html

[26] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *The primary-backup approach*. ACM Press, Addison-Wesley, 1993, pp. 199–216.

[27] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.